

# Automated Model Design using Genetic Algorithms and Model Checking

Raluca Lefticaru, Florentin Ipate, Cristina Tudose  
Department of Computer Science and Mathematics  
University of Pitesti

Str. Targu din Vale 1, 110040 Pitesti, Romania  
raluca.lefticaru@gmail.com, florentin.ipate@ifsoft.ro, cristina\_ferent@yahoo.com

**Abstract**—In recent years there has been a growing interest in applying metaheuristic search algorithms in model-checking. On the other hand, model checking has been used far less in other software engineering activities, such as model design and software testing. In this paper we propose an automated model design strategy, by integrating genetic algorithms (used for model generation) with model checking (used to evaluate the fitness, which takes into account the satisfied/unsatisfied specifications). Genetic programming is the process of evolving computer programs, by using a fitness value determined by the program’s ability to perform a given computational task. This evaluation is based on the output produced by the program for a set of training input samples. The consequence is that the evolved program can function well for the sample set used for training, but there is no guarantee that the program will behave properly for every possible input. Instead of training samples, in this paper we use a model checker, which verifies if the generated model satisfies the specifications. This approach is empirically evaluated for the generation of finite state-based models. Furthermore, the previous fitness function proposed in the literature, that takes into account only the number of unsatisfied specifications, presents plateaux and so does not offer a good guidance for the search. This paper proposes and evaluates the performance of a number of new fitness functions, which, by taking also into account the counterexamples provided by the model checker, improve the success rate of the genetic algorithm.

**Keywords**—model design; genetic algorithms; model checking; fitness function.

## I. INTRODUCTION

*Search-based software engineering (SBSE)* [1], [2] is a relatively new field, which uses the strength of metaheuristic search algorithms, like genetic algorithms, simulated annealing or particle swarm optimisation, to solve difficult problems from software engineering. Typically, these are complex problems, with large search-spaces, for which classical optimisation techniques cannot be applied. Recently, SBSE has had a fast development and *search-based software testing* [3] has been the most studied of its subdomains. Even though there has been a growing interest in applying heuristic and metaheuristic search algorithms in model-checking (especially to combat the state-explosion problem), model checking has been less used to solve other problems of SBSE, like model design and software testing.

Model checking [4] is a method of verifying if a simplified model of a system meets a given specification, expressed

as a temporal logic formula. Efficient symbolic algorithms have been used to explore (infinite) paths in the model and to check whether the specification holds or not. In recent years, a great interest has been shown in applying search techniques to solve model-checking problems. Several approaches, using heuristics [5] or genetic algorithms [6], [7], have been developed in order to face the state-explosion problem and to explore very large state spaces. Other metaheuristic search techniques (like ant colony optimisation [8]–[12], particle swarm optimisation [13]) have also been applied in model checking. These have been used to find liveness and safety properties violations, deadlocks and protocol errors in the systems. The results obtained by Alba and Chicano outperformed the state-of-the-art algorithms in model checking, which are exhaustive, deterministic and require huge amounts of computational resources if the checked model is large [8], [9], [12].

Related work regarding model generation using model checking includes the approaches of Johnson [14], Goldsby et al. [15], [16], which focus on state models generation. Katz and Peled employed genetic programming and model checking techniques to generate mutual exclusion algorithms [17], [18].

In this paper we propose an integration between automated model design and verification, by using genetic algorithms [19] for model generation and a model checking based fitness function. Traditionally, *genetic programming* [20] is used to develop computer programs and the fitness value is determined by the program’s ability to perform a given computational task. This evaluation is based on the program’s output for a set of training input samples. The consequence is that the evolved program can function well for the sample set used in training, but there is no guarantee that it will behave properly for every possible input.

Instead of training samples, we use a model checker, which verifies if the generated model satisfies the specifications. This approach is used by Johnson [14]; however, the fitness function proposed by Johnson, that takes into account only the number of unsatisfied specifications, presents plateaux and so does not offer a good guidance for the search. This paper proposes and evaluates the performance of a number of new fitness functions, which, by taking also into account the counterexamples provided by the model

checker, improve the success rate of the genetic algorithm.

The paper is structured as follows. Section II presents our model generation approach, the corresponding experimental results are given in Section III. Related work is mentioned in Section IV and the conclusions are drawn in Section V.

## II. MODEL GENERATION STRATEGY

In what follows we present our approach for automatic model generation from the information known about the system, expressed in the form of temporal logic specifications [4].

**Given:** An informal description of a system, the number of states and their labels  $S = \{s_1, \dots, s_n\}$ , other variables in the system  $V = \{v_1, \dots, v_m\}$  and their domains of values  $D = \{D_1, \dots, D_m\}$ , a set of LTL or CTL specifications  $\Phi = \{\phi_1, \dots, \phi_k\}$ .

**Problem:** Generate a (state machine) model of the system, which has the aforementioned states and variables and satisfies the properties expressed in temporal logic (e.g. safety and liveness properties).

The target is to obtain a model (i.e. the state transitions and the variable calculations for the next state), which satisfies, for every feasible path, all the constraints expressed by the temporal logic specifications.

Consider a finite state model, represented as a 4-tuple,  $M = (S, S_0, V, T)$ , where  $S = \{s_1, \dots, s_n\}$  is the set of states,  $S_0 \subseteq S$  is the initial set state (or  $s_0 \in S$  is the initial state),  $V = \{v_1, \dots, v_m\}$  is a set of variables (either monitored or control variables) having the finite domains  $D = \{D_1, \dots, D_m\}$  and  $T$  is a *transformation* relation, describing allowed state transitions. At each moment the system can be described by the current state and the values of its variables ( $state, v_1, \dots, v_m$ ). To capture the system dynamics/behaviour, a genetic algorithm [19] is employed to automatically specify, for every configuration of the system ( $state, v_1, \dots, v_m$ ), the next possible state(s) ( $state', v'_1, \dots, v'_m$ ). In the following examples only the deterministic case is illustrated (i.e. there is only one next state and so  $T$  is a function), but the algorithm can be easily adapted for the case in which  $T$  is a relation (there can be many next states).

In our experiments we have used the NuSMV model-checker [21]. NuSMV can check if the properties specified as temporal logic (CTL or LTL) formulas [4] are satisfied and return 'true' or 'false' and also a counterexample in the latter case. We have used both CTL and LTL specifications; when the specifications are equivalent, it is irrelevant which type is chosen. Furthermore, we have also used the bounded model checking (BMC) option of NuSMV; in this case, only LTL specification are accepted. For simplicity, in what follows only the LTL specifications will be provided.

### A. Chromosome representation

One can describe the transformation  $T$  by considering all possible combinations between the values of the current

state and variables, i.e.  $(state, v_1, \dots, v_m) \in S \times D_1 \times \dots \times D_m$ , and specifying in each case the next values ( $state', v'_1, \dots, v'_m$ ). Obviously, this is feasible when the system has only a few variables and their domains are finite and not too large. In this case, all possible system configurations ( $state, v_1, \dots, v_m$ ) can be obtained by simple computations, by considering all the combinations. When there are many variables and/or their domain range is large, a category-partition approach can be used instead, as illustrated in Section III-B. The problem is to determine, for each current state of the system, the corresponding next system configuration ( $state', v'_1, \dots, v'_m$ ), such that the given specifications  $\Phi = \{\phi_1, \dots, \phi_k\}$  hold.

A genetic algorithm can be used to automatically generate a completely specified deterministic system, by encoding in a chromosome the following information:

- the initial state;
- for each variable, the initial value;
- for the current state, the next state, taking into account all the possible combinations from  $S \times D_1 \times \dots \times D_m$
- for each variable, the next value of the variable, taking into account all the possible combinations from  $S \times D_1 \times \dots \times D_m$ .

To represent the aforementioned elements, we need a real encoded chromosome, with  $1 + m + (1 + m) \cdot (|S| \cdot |D_1| \cdot \dots \cdot |D_m|)$  genes. If the domains  $D_1, \dots, D_m$  are finite and  $D_i = \{d_{i_1}, \dots, d_{i_r}\}$ , for each variable  $v_i \in D_i$  it is sufficient to encode in the chromosome gene just the index  $j, j = 1..r$  of its corresponding value from the domain  $D_i$ . Having this real (or integer) representation of the system, a fitness function will evaluate the adequacy of each possible solution (individual or chromosome) according to the set of specifications  $\Phi = \{\phi_1, \dots, \phi_k\}$  given as input, as shown next.

### B. Fitness functions

In order to evaluate the fitness of each individual, we have proposed and tested several fitness functions, which analyze the response given by the model checker, as described below. The model checking based fitness function proposed by Johnson [14] counts the number of unsatisfied conditions from the specification set  $\Phi = \{\phi_1, \dots, \phi_k\}$ . Since this function uses only some discrete values, it has plateaux (for every value  $0, 1, \dots, k$ ) and, consequently, it does not offer a sufficient guidance for the search. Since search guidance is an essential part of a genetic algorithm, this paper focuses on improving this fitness function; several variants (functions  $f_2 - f_5$  below) are proposed and their performances are compared. The following fitness functions were used in our experiments:

- $f_1$  counts only the *unsatisfied specifications* [14]:  $f_1 = \sum_{i=1}^k \text{unsat}(\phi_i)$ , where  $\text{unsat}(\phi_i) = 1$  if  $\phi_i$  is false and 0 otherwise. This is the original function proposed by Johnson.

- $f_2 = \sum_{i=1}^k w_i \cdot \text{unsat}(\phi_i)$  is a weighted version of  $f_1$ , which uses a set of weights  $W = \{w_1, \dots, w_k\}$ . Empirically, we have determined that a set  $W$ , which assigns higher weights to the specifications that are 'simpler' (such as  $\phi_1, \phi_2$  from the Traffic lights example from Section III-A), provides a better guidance for the search.
- $f_3$  is a 'smoother' version of  $f_1$ , which adds, for each unsatisfied specification, the inverse value of a counterexample length:  $f_3 = \sum_{i, \text{unsat}(\phi_i)=1} (\text{unsat}(\phi_i) + 1/\text{lng}(c(\phi_i)))$ . Here,  $\text{lng}(c(\phi_i))$  represents the length of a counterexample returned by the model checker and it is computed by just counting the tokens (strings) that appear in the counterexample  $c(\phi_i)$ .
- $f_4$  takes into account the number of states that appear in the returned counterexample, i.e.  $f_4 = \sum_{i, \text{unsat}(\phi_i)=1} (\text{unsat}(\phi_i) + 1/\text{no\_states}(c(\phi_i)))$ , where  $\text{no\_states}$  is the number of system states or configurations that appear in the counterexample  $c(\phi_i)$ .
- $f_5$  is identical to  $f_4$ , except that the counterexample used in the calculation is always the *shortest*, as given by the BMC option of NuSMV, i.e.  $f_5 = \sum_{i, \text{unsat}(\phi_i)=1} (\text{unsat}(\phi_i) + 1/\text{no\_states}(c_{\min}(\phi_i)))$ , where  $c_{\min}(\phi_i)$  is the shortest counterexample for the false specification  $\phi_i$ .

### III. EXPERIMENTAL RESULTS

#### A. Traffic lights system

The first example considered is a traffic lights system, which can be described by a finite state machine having three states  $\{\text{red}, \text{green}, \text{yellow}\}$ . An informal description of the system is as follows: (a) After *red* and *green*, the traffic lights will become *yellow*. (b) If before *yellow* the lights were *red*, then the next state will be *green*, otherwise *red*. A set of specifications can be build, containing the properties that the system must satisfy, such as liveness properties (something good will eventually happen): "F(state = *green*)", or safety properties (something bad never happens): "G(state = *red*  $\rightarrow$  X(!(state = *green*)))".

In order to formally specify this model, the boolean variable *redBefore* is used to record if before *yellow* the traffic lights were *red* or *green* (see Fig. 1(a)).

Having an array containing the state values  $\{\text{red}, \text{green}, \text{yellow}\}$ , the set of variables  $V = \{v_1\} = \{\text{redBefore}\}$  and the variable domain  $\{\text{true}, \text{false}\}$  (or equivalent  $\{0, 1\}$ ) as input, all possible combinations of  $\text{state} \times v_1$  can be considered. The encoded chromosome will contain information regarding: the initial state, the initial value of the variable  $v_1$ , the next state and the next value of the variable  $v_1$  for each combination. Based on this chromosome encoding, a file representing a SMV specification, with the following LTL specifications  $\Phi = \{\phi_1, \dots, \phi_5\}$  is generated.

- $\phi_1$ : G (state = red  $\rightarrow$  X(state = yellow))
- $\phi_2$ : G (state = green  $\rightarrow$  X(state = yellow))
- $\phi_3$ : G (state = yellow  $\rightarrow$  X (state = red  
| state = green))
- $\phi_4$ : G (state = red  $\rightarrow$ (X (state = yellow)  
& X (X (state = green))))
- $\phi_5$ : G (state = green  $\rightarrow$  (X (state = yellow)  
& X (X (state = red))))

The model-checker NuSMV [21] parses the file and returns true or false and a counterexample for each LTL specification.

This chromosome encoding (14 integer genes) was used for each of the fitness functions mentioned in section II-B; the success rates after 100 runs are given in Table I. They show that  $f_2$  usually improves the success rate of  $f_1$  and, in most cases,  $f_3, f_4, f_5$  give even higher success rates.

The success rates and the average number of generations needed to evolve a correct model of the system depend also on the number of specifications. In order to smooth out the fitness landscape, some statements were added to  $\Phi$ . For example, 'G (state = red  $\rightarrow$  X(!(state = red)))' and 'G(state = red  $\rightarrow$  X(!(state = green)))' act as stages on the way to complete the specification  $\phi_1$ .

For the genetic algorithm implementation we used the JGAP framework [22] with the following settings: population size 20, maximum number of evolutions 20, elitist GA (preserves the best individual from each generation), selection realized using a best chromosomes selector, that keeps the top 0.8 individuals for the next generation, a mutation operator with the mutation rate 1/12. In order to improve the success rate of the genetic algorithm we used the following recombination operators: 1PX - one point crossover and UX - uniform crossover [19]. The search space contains  $3 \cdot 2 \cdot 3^6 \cdot 2^6 = 279936$  possible configurations and the problem has several solutions in this space.

#### B. Safety injection

The second experiment consisted in generating the control model for a simplified version of a control system for Safety Injection [23], [24]. This system monitors water pressure and injects coolant into a reactor core when the water pressure falls below some threshold. The model is inspired from [23]; we considered 6 variables, that monitor and control the system. *WaterPres*  $\in \{0, \dots, 50\}$ , *Block*  $\in \{\text{On}, \text{Off}\}$ , and *Reset*  $\in \{\text{On}, \text{Off}\}$  denote *monitored quantities*. *Block* and *Reset* are switches that can be turned 'On' or 'Off' by the system operator. The other three variables are a *mode class* *Pressure*  $\in \{\text{TooLow}, \text{Permitted}, \text{High}\}$ , a *term Overridden*  $\in \{\text{true}, \text{false}\}$  and *SafetyInjection*  $\in \{\text{On}, \text{Off}\}$  [23].

Obviously, enumerating all the possible combinations of all these variables is not feasible, due to their large number. One way to handle this issue is to use a *category partitioning* of variables. For example, in the

<pre> MODULE main VAR   state: {green, yellow, red};   redBefore : boolean; ASSIGN init(state) := red; init(redBefore) := 1; next(state) := case   state = red : yellow;    state = yellow &amp; !redBefore : red;   state = yellow &amp; redBefore : green;   state = green : yellow;  esac; next(redBefore) := case   state = red : 1;    state = green : 0;   1 : redBefore; esac; </pre> <p style="text-align: center;"><b>(a)</b></p>	<pre> MODULE main VAR   state: {green, yellow, red};   v1 : boolean; ASSIGN init(state) := green; init(v1) := 0; next(state) := case   state = red &amp; v1 : yellow;   state = red &amp; !v1 : red;   state = yellow &amp; !v1 : green;   state = yellow &amp; v1 : red;   state = green &amp; v1 : red;   state = green &amp; !v1 : yellow;  esac; next(v1) := case   state = red &amp; v1 : 0;   state = red &amp; !v1 : 1;   state = yellow &amp; !v1 : 0;   state = yellow &amp; v1 : 1;   state = green &amp; v1 : 1;   state = green &amp; !v1 : 1; esac; </pre> <p style="text-align: center;"><b>(b)</b></p>	<pre> (state, redBefore)   (red, 1)   ↓   (yellow, 1)   ↓   (green, 1)   ↓   (yellow, 0)   ↓   (red, 0)   ↓   (yellow, 1)   ↓   (green, 1)   ↓   (yellow, 0)   ↓   (red, 0)   ↓   (yellow, 1)   ↓   ... </pre> <p style="text-align: center;"><b>(c)</b></p>	<pre> (state, v1)   (green, 0)   ↓   (yellow, 1)   ↓   (red, 1)   ↓   (yellow, 0)   ↓   (green, 0)   ↓   (yellow, 1)   ↓   (red, 1)   ↓   (yellow, 0)   ↓   (green, 0)   ↓   (yellow, 1)   ↓   ... </pre> <p style="text-align: center;"><b>(d)</b></p>
--	---	--	---

Figure 1. Traffic lights model: (a) Designed specification, (b) Automatically generated specification, (c) Path in designed model, (d) Path in generated model

case  $WaterPres \in \{0, \dots, 50\}$  the domain  $\{0, \dots, 50\}$  was split into  $\{0, \dots, Low - 1\}, \{Low, \dots, Permit - 1\}, \{Permit, \dots, 50\}$ , where  $Low = 15$  and  $Permit = 25$  are constants. The predicates used for category partitioning are inspired from the specifications given as input. Also, in order to simplify the problem, it is useful to determine how variables depend on one another (this is because, if it is known that a variable depends only on a subset of the set of variables  $V$ , the corresponding combination will take into account the subset and not the whole set  $V$ ).

For this system, the genetic algorithm aimed to determine only the control transitions, all the information regarding the monitored variables having been already known. The specifications used were inspired from [24] and 2 equivalent groups were used: one with only 16 specifications and another with 30 LTL specifications. The search space dimension was  $3^3 \cdot 2^7 \cdot 2^6 = 221184$ , with only one solution to the problem. The success rates obtained, for the same genetic algorithm configuration as in the previous model, are shown in Table I. The results are lower than those for the Traffic Lights system; this is explained by the fact that the problem has only one solution in the search space.

### C. Vehicle

The last experiment consisted in modelling a vehicle that moves on the perimeter of a square. The system has 4 states,

$\{left\_up, left\_down, right\_up, right\_down\}$  and can accept 5 commands  $\{go\_up, go\_down, go\_left, go\_right, do\_nothing\}$ . In order to obtain a completely specified finite state machine, we considered all the  $(state, command)$  combinations. The next state, represented as a gene in the chromosome, was specified for each combination  $(state, command)$ , where  $command$  represents an input variable to the system. Thus, the search space for the genetic algorithm has the size  $4^{20}$  and the problem has only one solution. For this reason, the maximum allowed number of evolutions was set to 200. We used a set of 20 LTL specifications, which described each transition of the system. As these specifications had the same complexity, there was no reason to use weights and the function  $f_2$ . The results, summarized after 20 runs of the genetic algorithm, are shown in Table I.

### D. Remarks

The conclusions drawn from these 3 experiments are:

- The fitness functions  $f_2 - f_5$  improve the success rate of the genetic algorithm, compared to  $f_1$ .
- The weights used in the  $f_2$  formula have a great impact on its success. When higher weights are assigned to simpler specifications and lower weights to more complex specifications, the results obtained are better than in the opposite case. For example, in the Traffic lights system, having the specifications  $\phi_1, \dots, \phi_5$  from

Table I  
SUCCESS RATES AND AVERAGE NUMBER OF GENERATIONS (FOR SUCCESSFUL RUNS)

Problem	Number of specifications	Crossover operator	Fitness functions									
			$f_1$		$f_2$		$f_3$		$f_4$		$f_5$	
Traffic lights	5	1PX	64%	5.57	88%	7.09	93%	6.07	98%	5.95	98%	6.51
		UX	77%	5.90	89%	5.82	94%	5.20	95%	6.01	98%	5.31
	17	1PX	90%	6.24	91%	7.09	95%	5.82	96%	6.03	98%	4.84
		UX	99%	5.64	91%	5.31	99%	5.46	99%	4.73	100%	4.35
Safety injection	16	1PX	40%	11.73	74%	13.10	54%	12.44	58%	12.55	80%	14.25
		UX	57%	9.65	88%	11.85	73%	11.04	71%	11.45	80%	13.12
	30	1PX	40%	9.75	48%	9.77	84%	9.57	80%	11.0	90%	9.11
		UX	74%	7.03	72%	7.29	84%	8.31	86%	7.98	100%	8.6
Vehicle	20	1PX	100%	58.15	-	-	100%	54.35	100%	56.65	100%	51.2
		UX	100%	41.31	-	-	100%	44.36	100%	48.6	100%	38

section III-A, the set of weights  $W = \{2, 2, 2, 1, 1\}$  produced 88-91% success rates (see Table I), compared with the set  $W = \{1, 1, 1, 2, 2\}$ , for which the success rate was only 54%<sup>1</sup>.

- An increased number of (not very complex) LTL specifications helps the search.
- The UX operator is more appropriate than 1PX for this kind of problems.
- The fitness function  $f_5$ , although very accurate (using the shortest counterexample) and providing in most cases the highest success rates, is by far the slowest to compute (approximately up to 10 times much slower than  $f_1 - f_4$ , when the maximum number of iterations for the BMC option of NuSMV was set to 10).

#### IV. RELATED WORK

Johnson [14] proposes model checking as a fitness measure, with an accumulative evolution strategy, and uses this for model generation of a control system for coffee vending machines. The system from this case study is modelled using communicating finite state machines as description language and takes the following information as input: list of labels for nodes, list of variables and channels, one communicating state machine that models the users behaviour and a list of CTL statements. However, the fitness function proposed uses only the number of unsatisfied CTL conditions.

An approach to generate behavioural models (UML state diagrams) using *digital evolution* is presented in [15], [16]. The state machines obtained have to support some *key scenarios*, satisfy critical *properties*, specified by the developer, extend existing behavioural models and meet some software engineering *metrics*. The research reported in [15], [16] is realized using AVIDA, a digital evolution platform, that works with a population of self-replicating organisms, which is subject to mutations and natural selection. The *merit* (fitness) of an organism is assessed by the tasks it can

perform correctly. In this case, the tasks were represented by key scenarios, properties satisfied, software metrics.

In their experiments, Goldsby et al. [15], [16] used information regarding some classes of a Robot Navigation System (RNS), some state diagrams that described a RNS able to navigate to a destination, but could not avoid obstacles. The requirement was to find a behavioural model that satisfied two new scenarios (with an obstacle and without an obstacle), two properties (it had to avoid obstacles and reach its destination). Several models that comply with these requirements can be built and consequently software engineering metrics were also used: minimum number of transitions (rewarding an organism for generating state diagrams with fewer transitions than the rest of the population) and determinism (rewarding an organism for generating state diagrams that are deterministic).

Lai et al. present in [25] the application of genetic algorithms to the problem of learning a regular set from queries and counterexamples [26]. They use adaptive model checking in order to iteratively construct a state machine model of the system. The fitness function counts the number of training examples, correctly classified by a deterministic finite state automaton.

#### V. CONCLUSIONS AND FUTURE WORK

This paper addresses the problem of automatic model generation, using genetic algorithms. Its main contribution is to propose and empirically evaluate a new type of fitness function, which extends the functions already used in the literature. The fitness functions proposed use the number of unsatisfied specifications and a measure of the counterexample length, for each unsatisfied specification. The advantages of this type of fitness function are: its form is general and easy to compute, the function improves the success rate of the genetic algorithm and works well with a uniform crossover operator.

Future work concerns the application of this kind of fitness function to generate more complex models (and find possible solution to the state explosion problem), finding tuning options that improve the genetic algorithms as well as other

<sup>1</sup>In Table I, for the fitness function  $f_2$ , only the best results, corresponding to the set of weights assigning higher values to simpler specifications and lower weights to more complex specifications, are shown.

optimisation algorithms. Other types of fitness functions, that use additional information from the result of the model checker, may also be investigated.

#### ACKNOWLEDGMENT

The work of Raluca Lefticaru and Florentin Ipate is supported by the CNCSIS grant IDEI 496/2009, *An integrated evolutionary approach to formal modelling and testing (EvoMT)*. The authors are grateful to reviewers for their comments.

#### REFERENCES

- [1] M. Harman and B. F. Jones, "Search-based software engineering," *Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
- [2] M. Harman, "The current state and future of search based software engineering," in *Proceedings of International Conference on Software Engineering / Future of Software Engineering 2007 (ICSE/FOSE '07)*. IEEE Computer Society, 2007, pp. 342–357.
- [3] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
- [4] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.
- [5] A. Groce and W. Visser, "Heuristics for model checking Java programs," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 4, pp. 260–276, 2004.
- [6] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 117–127, 2004.
- [7] E. Alba, F. Chicano, M. Ferreira, and J. A. Gomez-Pulido, "Finding deadlocks in large concurrent Java programs using genetic algorithms," in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO '08)*. ACM, 2008, pp. 1735–1742.
- [8] E. Alba and F. Chicano, "Ant colony optimization for model checking," in *Proceedings of the 11th International Conference on Computer Aided Systems Theory (EUROCAST 2007)*, vol. 4739. Springer, 2007, pp. 523–530.
- [9] —, "Finding safety errors with ACO," in *GECCO '07*. ACM, 2007, pp. 1066–1073.
- [10] —, "Searching for liveness property violations in concurrent systems with ACO," in *GECCO '08*. ACM, 2008, pp. 1727–1734.
- [11] F. Chicano and E. Alba, "Finding liveness errors with ACO," in *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI '08)*. IEEE Computer Society, 2008, pp. 3002–3009.
- [12] —, "Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models," *Information Processing Letters*, vol. 106, no. 6, pp. 221–231, 2008.
- [13] M. Ferreira, F. Chicano, E. Alba, and J. A. Gomez-Pulido, "Detecting protocol errors using particle swarm optimization with Java Pathfinder," in *Proceedings of the High Performance Computing & Simulation Conference*, 2008, pp. 319–325.
- [14] C. Johnson, "Genetic programming with fitness based on model checking," in *Proceedings of the 10th European Conference on Genetic Programming (EuroGP 2007)*, vol. 4445. Springer, 2007, pp. 114–124.
- [15] H. J. Goldsby, B. H. Cheng, P. K. McKinley, D. B. Knoester, and C. A. Ofria, "Digital evolution of behavioral models for autonomic systems," in *Proceedings of the 2008 International Conference on Autonomic Computing*. IEEE Computer Society, 2008, pp. 87–96.
- [16] H. Goldsby and B. H. C. Cheng, "Avida-MDE: a digital evolution approach to generating models of adaptive software behavior," in *GECCO '08*. ACM, 2008, pp. 1751–1758.
- [17] G. Katz and D. Peled, "Model checking-based genetic programming with an application to mutual exclusion," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 2008, pp. 141–156.
- [18] —, "Genetic programming and model checking: Synthesizing new mutual exclusion algorithms," in *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA '08)*, vol. 5311. Springer, 2008, pp. 33–47.
- [19] Z. Michalewicz, *Genetic Algorithms Plus Data Structures Equals Evolution Programs*. Springer-Verlag, 1996.
- [20] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, 1992.
- [21] NuSMV model checker (homepage). [Online]. Available: <http://nusmv.irst.itc.it/>
- [22] K. Meffert et al. JGAP - Java Genetic Algorithms and Genetic Programming Package. [Online]. Available: <http://jgap.sf.net>
- [23] R. Bharadwaj and C. L. Heitmeyer, "Model checking complete requirements specifications using abstraction," *Autom. Softw. Eng.*, vol. 6, no. 1, pp. 37–68, 1999.
- [24] P. Ammann and P. E. Black, "A specification-based coverage metric to evaluate test sets," in *The 4th IEEE International Symposium on High-Assurance Systems Engineering (HASE '99)*. IEEE Computer Society, 1999, pp. 239–248.
- [25] Z. Lai, S. C. Cheung, and Y. Jiang, "Dynamic model learning using genetic algorithm under adaptive model checking framework," in *Proceedings of the Sixth International Conference on Quality Software (QSIC '06)*. IEEE Computer Society, 2006, pp. 410–417.
- [26] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.